Cookpad Summer Internship 2023
～ サーバーサイド編 ～

cookpad

今日のゴール

- クックパッドで使っている技術を理解する
  - BFF、GraphQL について
  - どうしてその技術を使っているのか
- Ruby on Rails のコードを見て、どこをどう読めばいいかわかるようになる
- 楽しいサマーインターンにする！

タイムテーブル

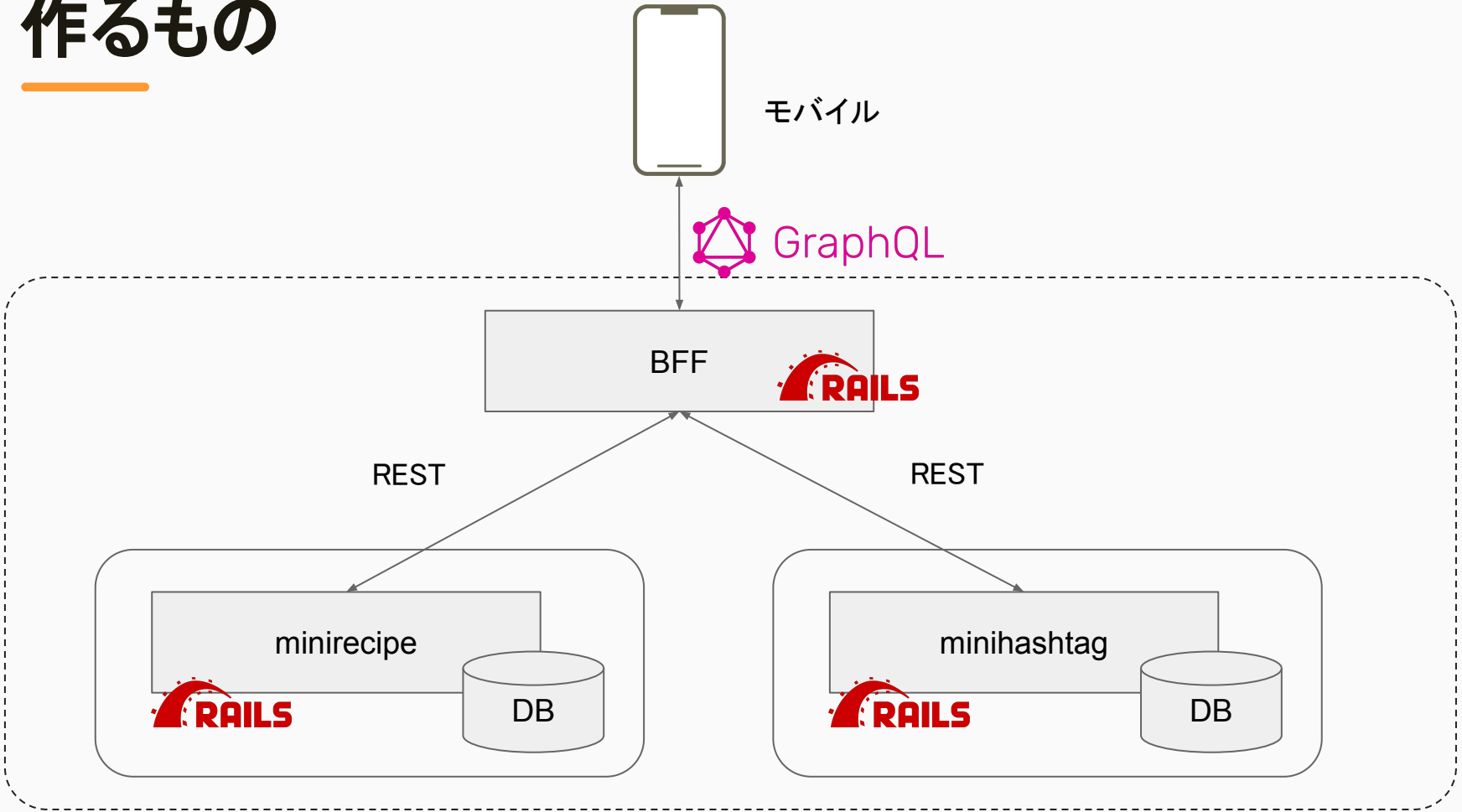
- 10:00 ~ 12:00 講義
- 12:00 ~ 13:00 お昼
- 13:00 ~ 15:00 ハンズオン
- 15:00 ~ 17:30 課題
- 17:30 ~ 18:00 まとめタイム

作るもの

iOS の講義で使った API を実際に作ります

- minirecipe
  - レシピ一覧・詳細
- minihashtag
  - ハッシュタグ投稿
- 2つのサービスを束ねる API（BFF）
  - クライアントからのリクエスト先を1つにします
  - 2つのサービスにリクエストを投げるのはややこしかったはず

作るもの


モバイル
GraphQL
BFF
RAILS
REST
REST
minirecipe
RAILS
DB
minihashtag
RAILS
DB

講義で扱う技術要素

- BFF
- GraphQL
- Ruby
- Rails

BFF

cookpad

BFFって何？

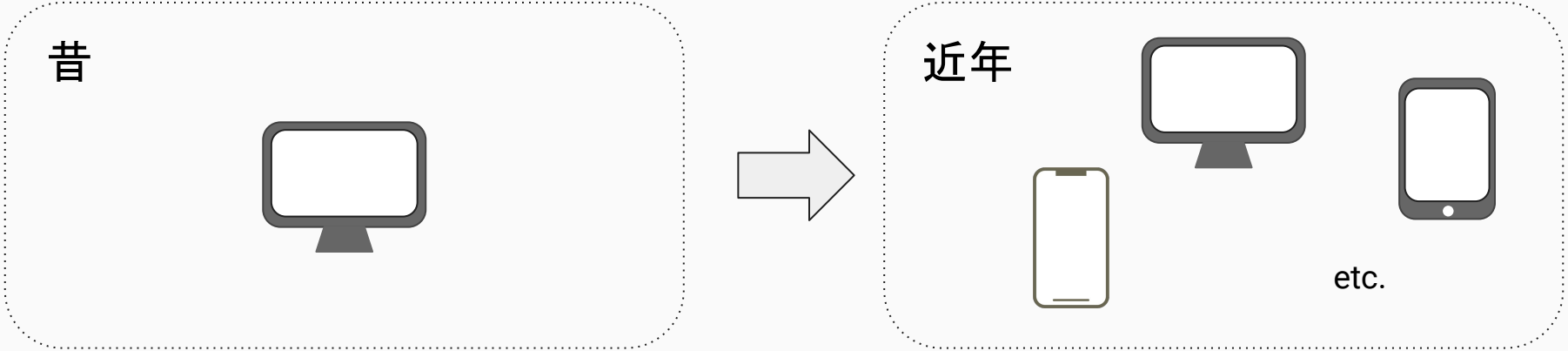
の前に……

近年の開発の難しさ

- 近年の開発は複雑さを増している
- ユーザー環境の多様化
  - いろんなクライアントがある（アプリ、Web etc…）
- アーキテクチャの複雑化
  - マイクロサービス
    - 1つのアプリケーションではなく複数のアプリケーションに分割して開発するようになった

ユーザー環境の多様化

- デスクトップ PC が主流だった時代 → モバイルも出てきた
- クライアントによって特性が異なる
  - PC Web：画面が大きくて、安定したネットワーク通信が可能
  - モバイル（アプリ、Web）：画面が小さくて、ハードウェアリソースに制限
- 利用シーンや UI も違う



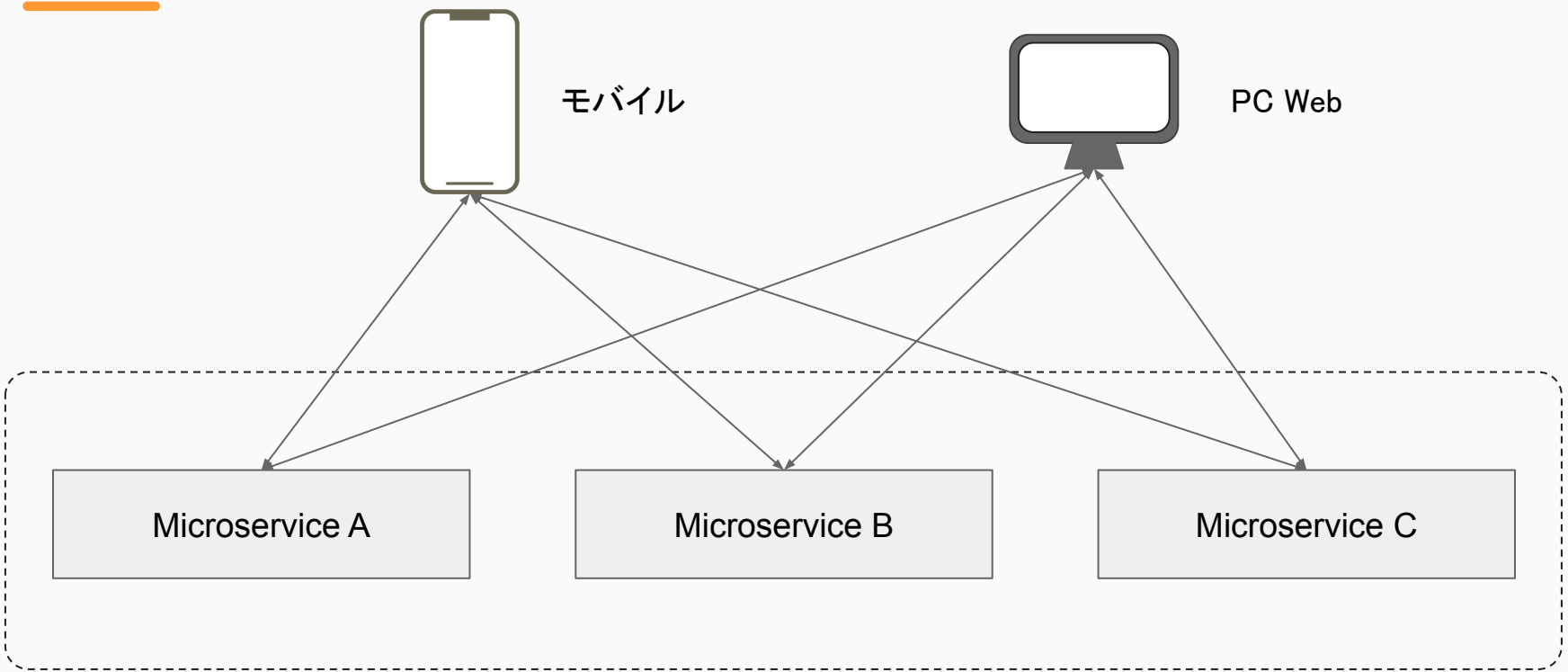
ユーザー環境の多様化

- 全種類のクライアントに対して、出し分けをする汎用 API を作るのは大変
  - 各種クライアントに対応するためのロジックの複雑化
  - 影響範囲の肥大化
    - 共有しているコードを変更すると各種クライアントにも影響がある
    - 複数のチームに影響のある変更だと
      - チームをまたがった調整が必要になる
      - コミュニケーションが大変になる

アーキテクチャの複雑化

- モノリス → マイクロサービス
  - 1つのアプリケーションではなく複数のアプリケーションに分割して開発するようになった
- クライアントから見たときの複雑さ
  - たくさんのマイクロサービス
    - どのサービスがどういうインタフェースを持っているのか
    - 複数サービスに直接にリクエストする場合はラウンドトリップタイムの問題も
  - 各マイクロサービスから得たデータの集約
    - 例: iOS 講義でやったレシピに紐づくハッシュタグの取得、レシピとの結合ロジック

クライアントから直接通信

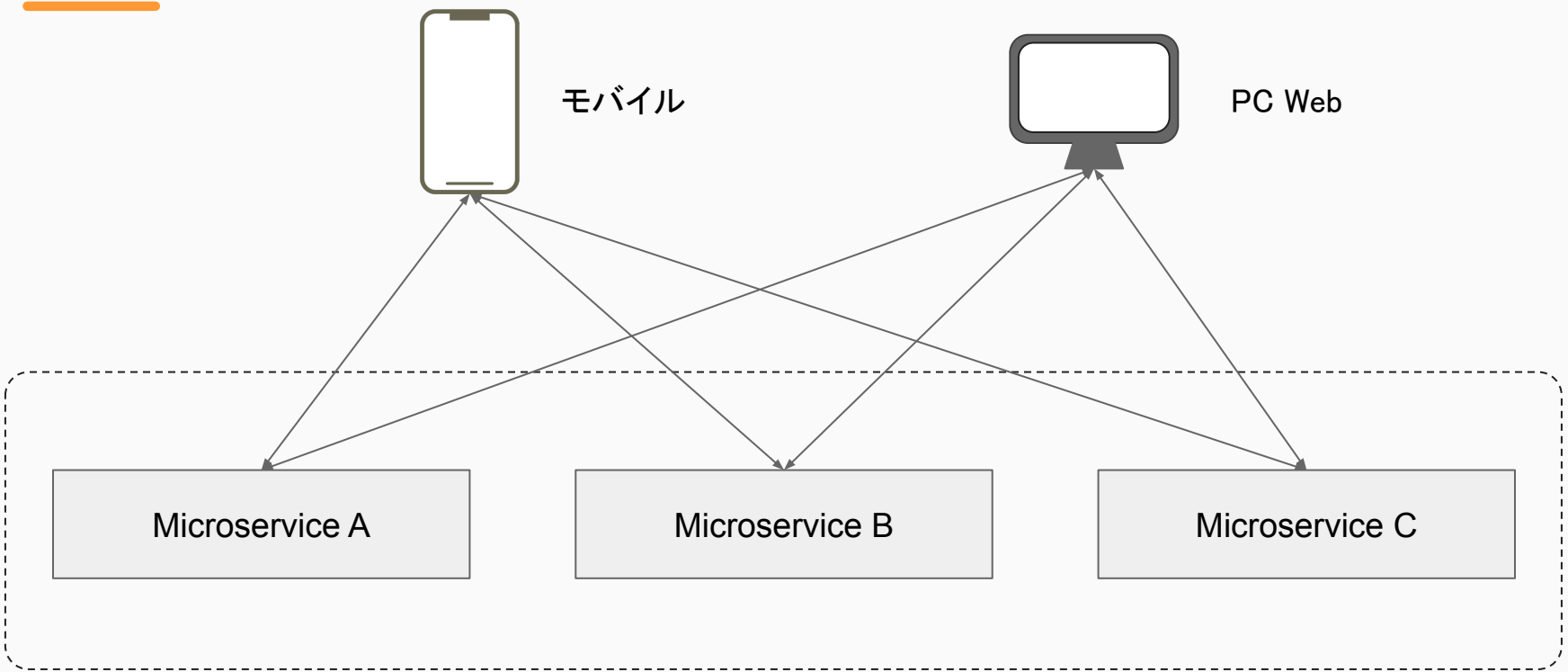


お互いにむずい

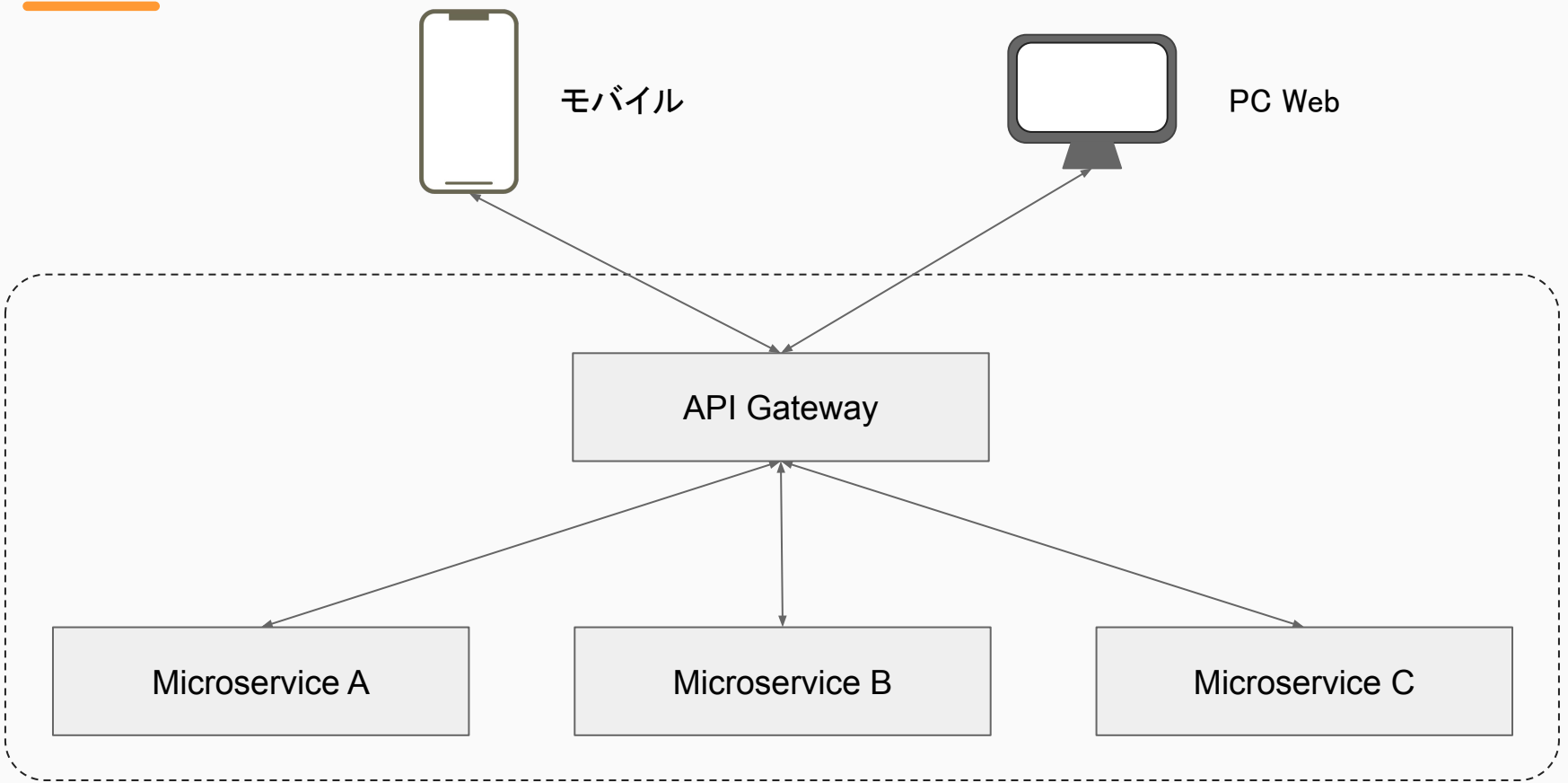
BFF (Backend for Frontend) とは

- 設計パターンの一つ
- フロントエンドと各種マイクロサービスとの間に位置し、フロントエンドに返すレスポンスを組み立てる
  - 各マイクロサービスにリクエストをする
  - 各マイクロサービスのレスポンスを結合
- バックエンドとフロントエンドの複雑性を排除する
- API Gateway という設計パターンの特別な形

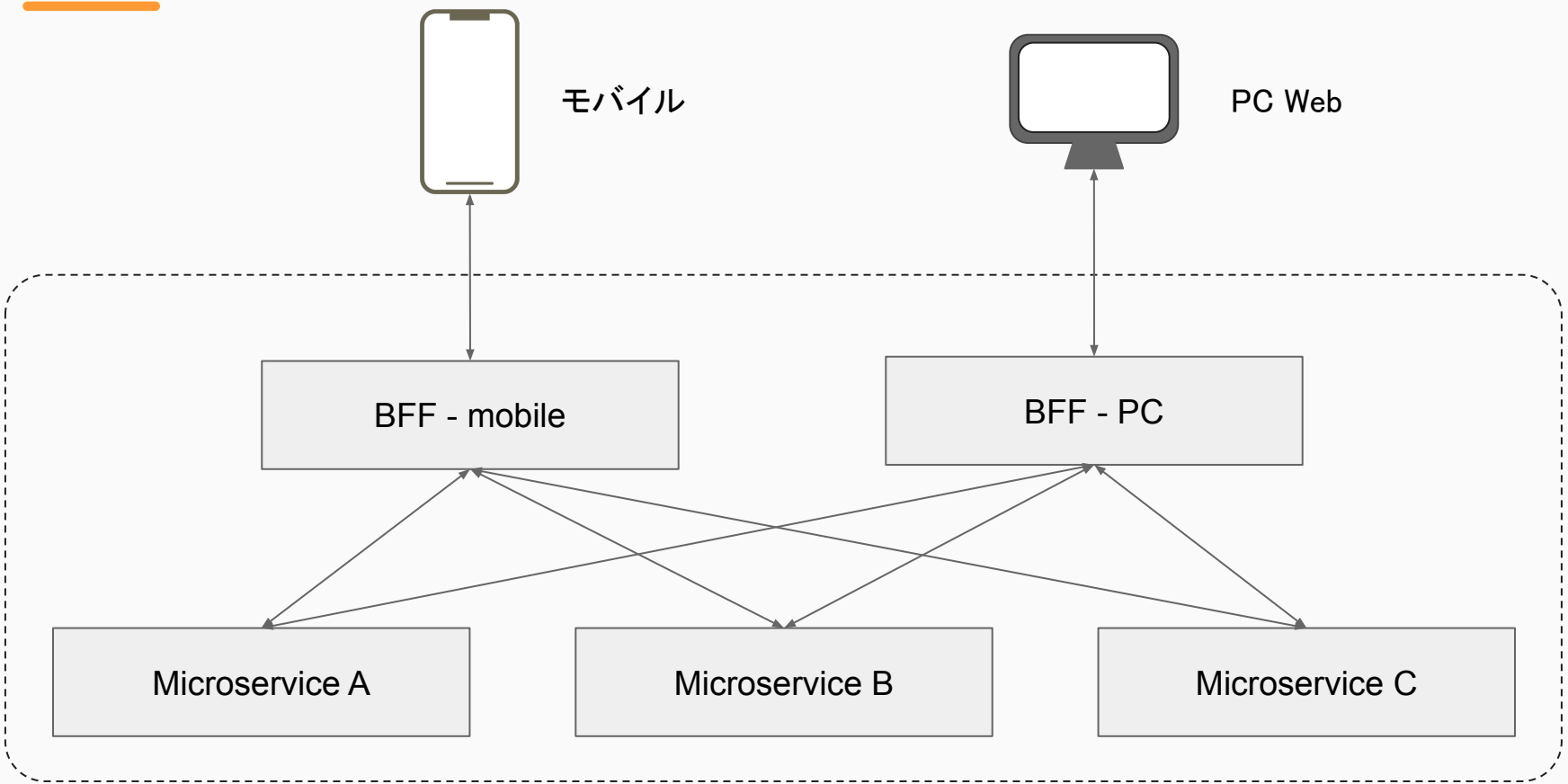
クライアントから直接通信（再）



API Gateway pattern

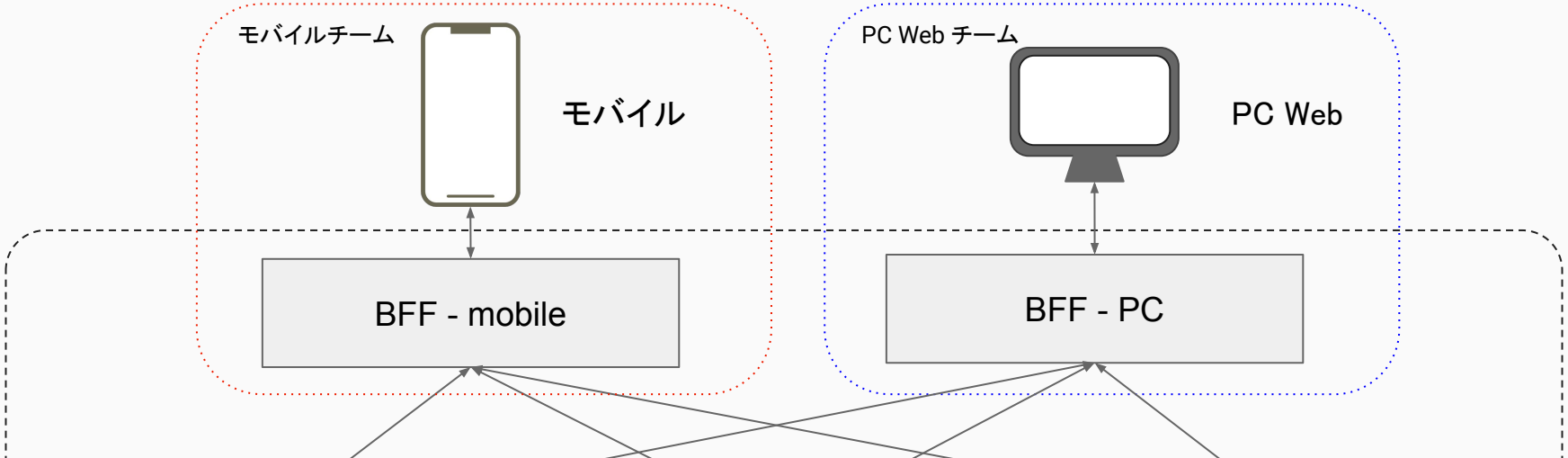


BFF


モバイル
PC Web
BFF - mobile
BFF - PC
Microservice A
Microservice B
Microservice C

BFF

- 1種類のクライアントに特化した層
  - BFF が各種サービスにリクエストを投げて、情報を集約してクライアントに返す
  - クライアントのユースケースに特化したレスポンスを組み立てる
- クライアント・バックエンドの差異を吸収



BFF - メリット

- クライアントは裏側のマイクロサービスを意識しないで済む
  - リクエスト先が BFF のみだけで済む
  - 1回のリクエストすればいい
  - マイクロサービスのリプレイスや仕様変更がしやすい
- クライアント・バックエンドの責務が明確になる
  - クライアントは BFF から受け取ったデータを表示する
  - バックエンドは要求されたデータを BFF に返す
  - BFF がバックエンドからデータを集めて加工し、クライアントが求める形で提供する
- クライアント特化、ユースケース特化なので、汎用性を考える必要がない
- その他様々なメリット（認証、キャッシュ、etc. ）

BFF - デメリット

- 管理するアプリケーションが増える
  - アプリケーションが多いとそれだけで大変
  - クライアントを開発するチームが管理する場合は、
    不慣れなサーバーサイド（BFF）を管理することになる
- ロジックが重複する
  - 複数の BFF で似たようなロジックを書くことになる
    - 単純なリソースを返すだけなど、意外と共通化できる部分は多くある
- 上手くやらないと BFF がすぐに肥大化する
  - 結合ポイントなので何でも書けちゃう。

クックパッドにおける BFF

- クックパッドもマイクロサービスを採用している
  - レシピ関連のマイクロサービスだけで20個ぐらいある
- クライアントから叩きたくないので BFF を運用している

クックパッドにおける BFF

- Orcha
  - モバイルアプリ (iOS, Android) 向け BFF
  - Java 製
  - https://techlife.cookpad.com/entry/2019-orcha-bff
- next-cookpad-api
  - スマートフォン Web 向け BFF
  - node.js 製
  - https://techlife.cookpad.com/entry/2020/12/01/093000

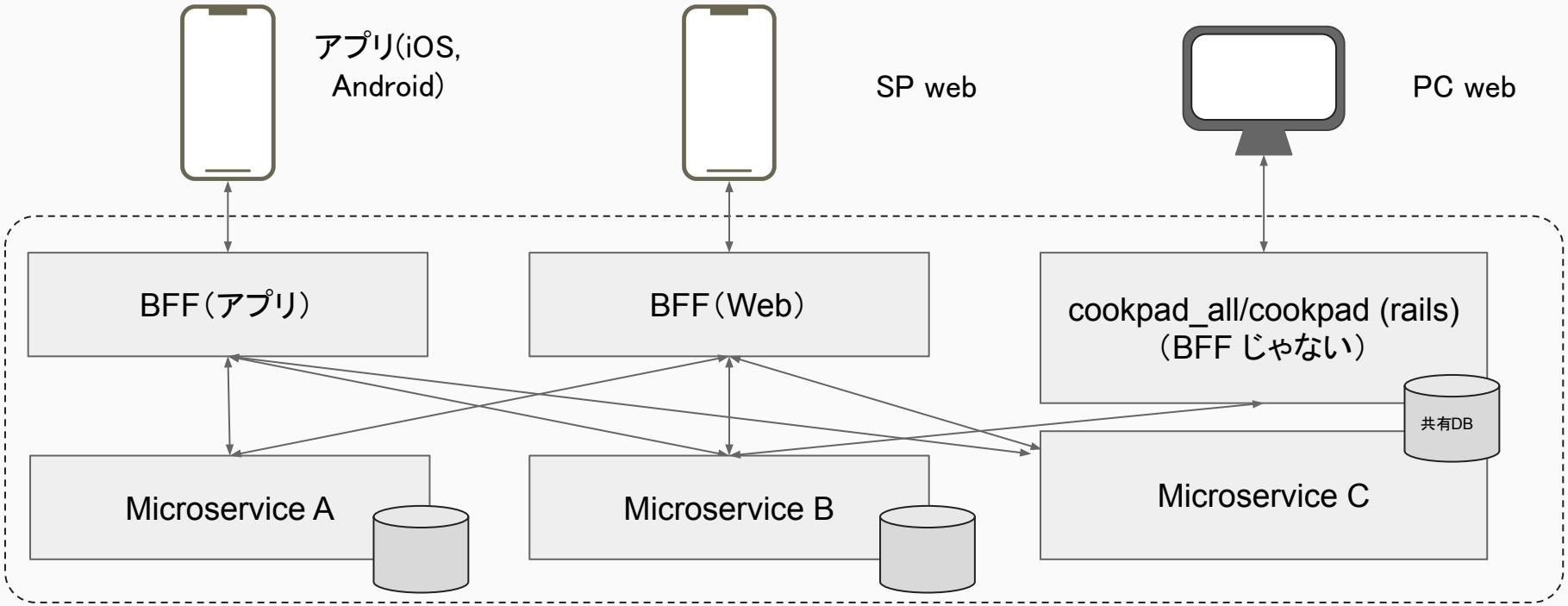
BFF 導入背景

- Orcha
  - iOS アプリ向けの新規機能を検証したい
  - 新規サービスとして追加
    - 検証なので撤退の可能性あり
    - 既存サービスは巨大で修正コスト高
  - iOS からのリクエスト回数を増やしたくない
- next-cookpad-api
  - Web Frontend モダン化プロジェクト
  - API 経由で各種リソースにアクセスするようになった
  - 複数のサービスを呼び分けるのは大変なのでBFF を用意

Orcha: https://techlife.cookpad.com/entry/2019-orcha-bff
next-cookpad-api: https://techlife.cookpad.com/entry/2020/12/01/093000

BFF - クックパッドの場合

アプリ、SPweb に BFF が一つずつ。PCweb はシンプルな Rails で別サービスも呼んでいる



BFF 導入後

- Orcha
  - アプリ向けの BFF としてアプリ開発の生産性は向上した
  - アプリエンジニアとサーバーサイドエンジニア両方が開発
    - アプリエンジニアには比較的親しみやすい
    - Rails エンジニアは Java に慣れておらず苦戦しがち
- next-cookpad-api
  - 開発が活発な画面から移行中
    - モダンな環境で快適な開発体験（型がある・脱jQuery）

突然ですが

抜き打ちクイズをします

BFF 理解度クイズ①

BFF は API Gateway という設計パターンの一種であり、

各プラットフォームごとに異なる BFF を用意するものである。

マルかバツか。

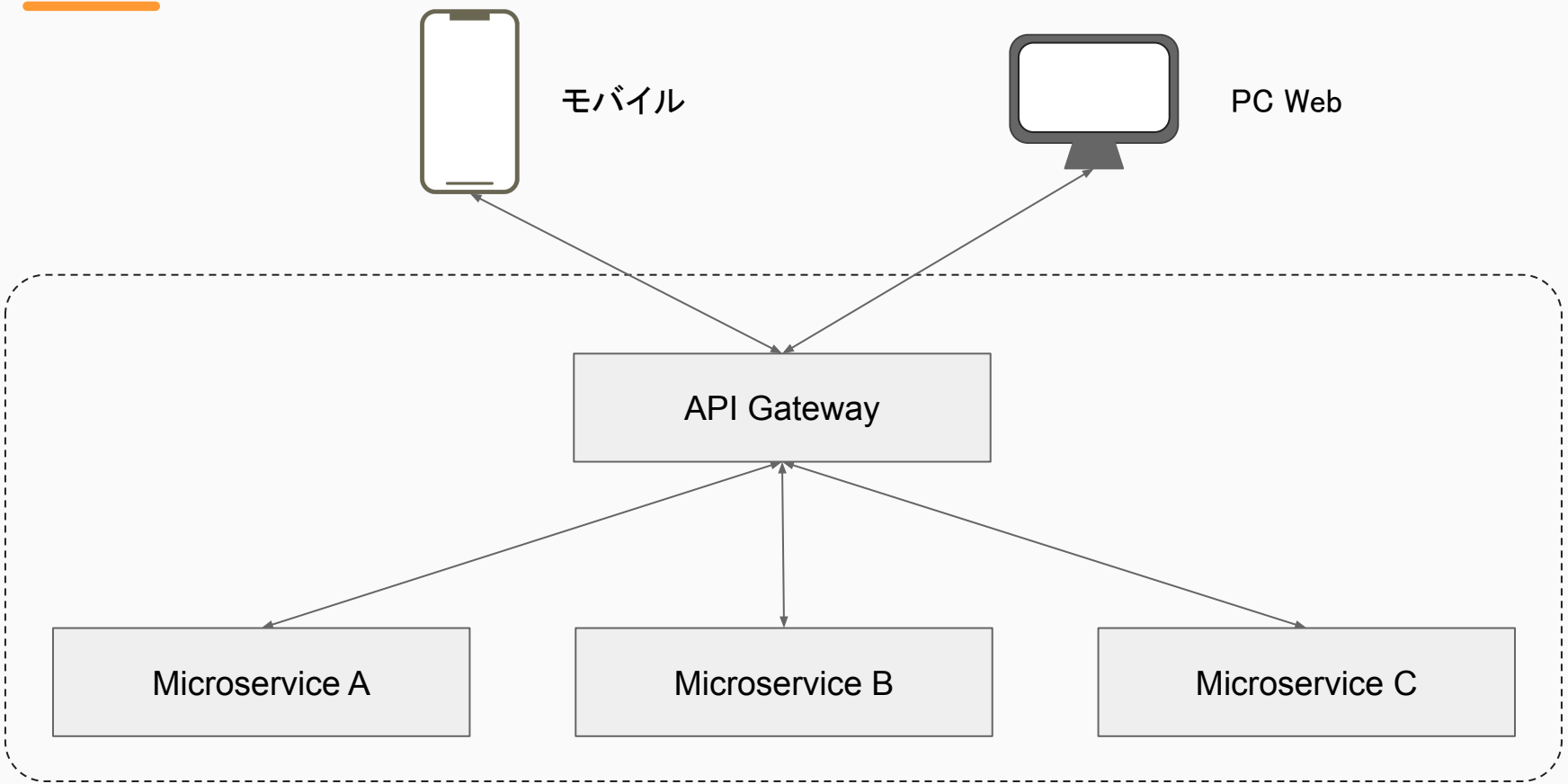
BFF 理解度クイズ①（解答）

マル

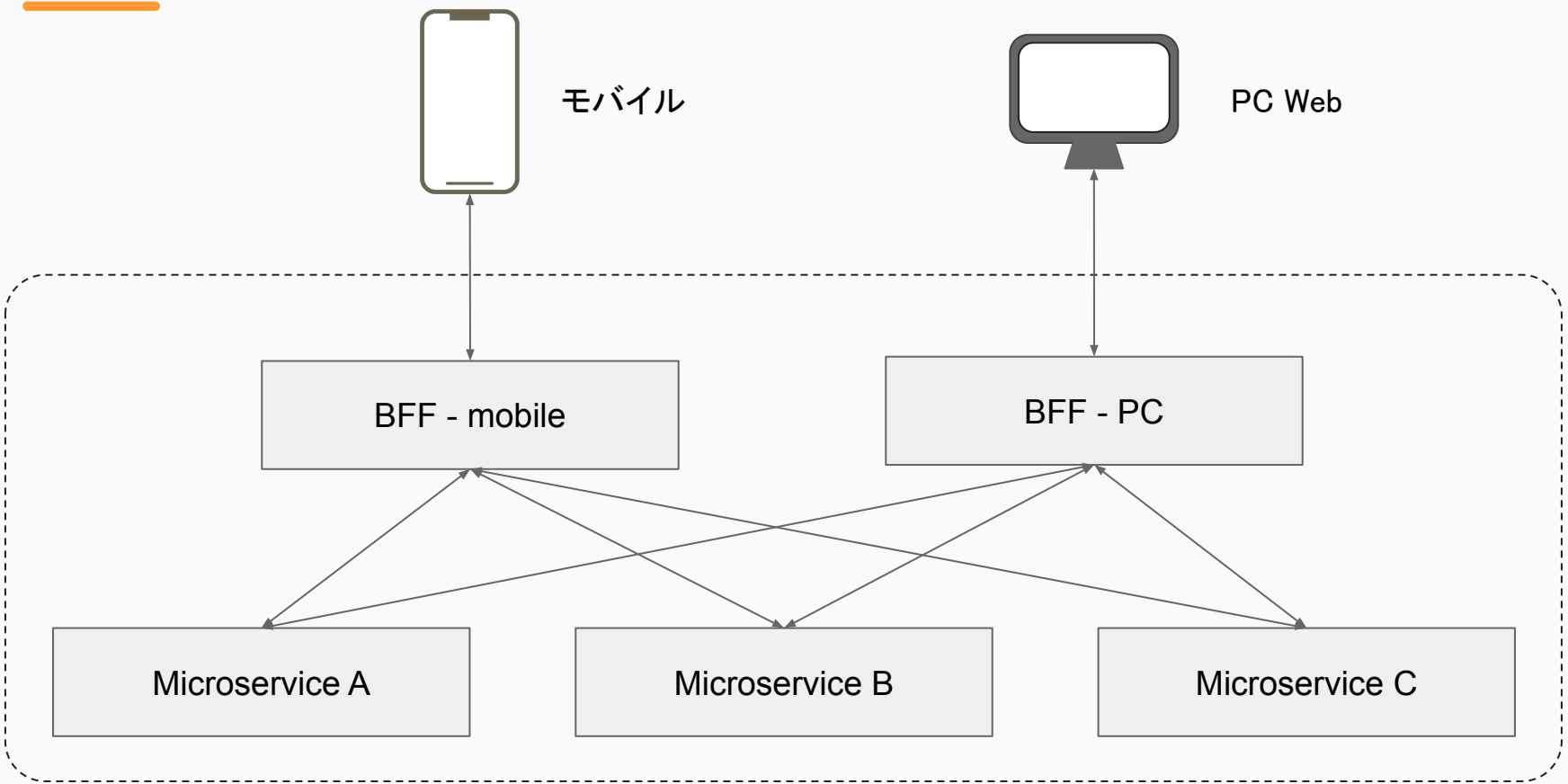
BFF は API Gateway という設計パターンの一種であり、

各プラットフォームごとに異なる BFF を用意するものです。

API Gateway pattern（再掲）



BFF（再掲）


モバイル
PC Web
BFF - mobile
BFF - PC
Microservice A
Microservice B
Microservice C

BFF 理解度クイズ②

BFF のデメリットの１つとして、

クライアントとサーバーの責務が曖昧になってしまうというものがある。

マルかバツか。

BFF 理解度クイズ②（解答）

バツ

通常、BFF を用いることで結合のロジックなどを
クライアントから BFF に移譲でき、
サーバー・クライアントの責務を明確にさせることができます。

BFF まとめ

- クライアントとマイクロサービスの間にある層
- 各マイクロサービスにリクエストをして、そのレスポンスをまとめる
- メリット
  - クライアントは裏側のマイクロサービスを意識しないで済む
  - クライアント・バックエンドの責務が明確になる
  - クライアント特化、ユースケース特化なので、汎用性を考える必要がない
- デメリット
  - 管理するアプリケーションが増える
  - ロジックが重複する
  - 上手くやらないとBFF がすぐに肥大化する
- クックパッドでもアプリ・Web に向けて BFF を採用している。

BFF は良いこと
ばっかり？

BFF のつらさ

- 実装が肥大化
  - 様々なロジック・データが BFF に集まっている
  - キャンペーンデータ、ユーザー状態に基づく出し分け、コンテンツの差し込み…
- Orcha と next-cookpad-api で技術スタックが大きく異なるのでコンテキストスイッチが大きい
    - Java x REST ↔ Node.js x GraphQL

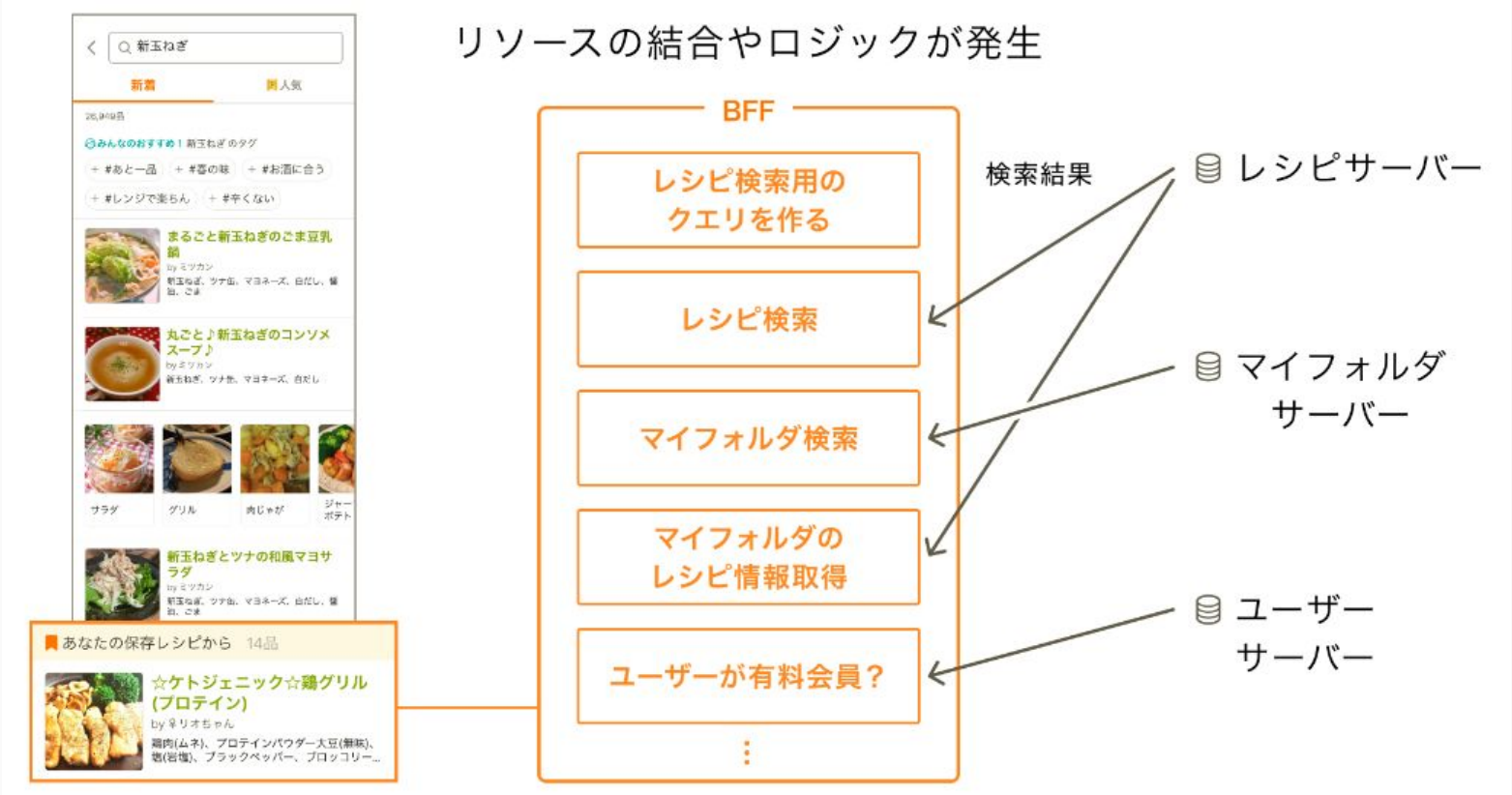
BFF のつらさ


新玉ねぎ
新着
人気
みんなのおすすめ！新玉ねぎのタグ
＋ #あと一品
＋ #春の味
＋ #お酒に合う
＋ #レンジで楽ちん
＋ #辛くない
差し込み
まるごと新玉ねぎのごま豆乳鍋
丸ごと♪新玉ねぎのコンソメスープ♪
差し込み
サラダ
グリル
肉じゃが
あなたの保存レシピから 14品
☆ケトジェニック☆鶏グリル(プロテイン)
差し込み

BFF のつらさ


新玉ねぎ
新着
人気
みんなのおすすめ！新玉ねぎのタグ
＋ #あと一品
＋ #春の味
＋ #お酒に合う
＋ #レンジで楽ちん
＋ #辛くない
まるごと新玉ねぎのごま豆乳鍋
丸ごと♪新玉ねぎのコンソメスープ♪
サラダ
グリル
肉じゃが
あなたの保存レシピから 14品
☆ケトジェニック☆鶏グリル(プロテイン)
by ☆リオちゃん
ハッシュタグサーバー
レシピサーバー
レシピサーバー
レシピサーバー
マイフォルダサーバー

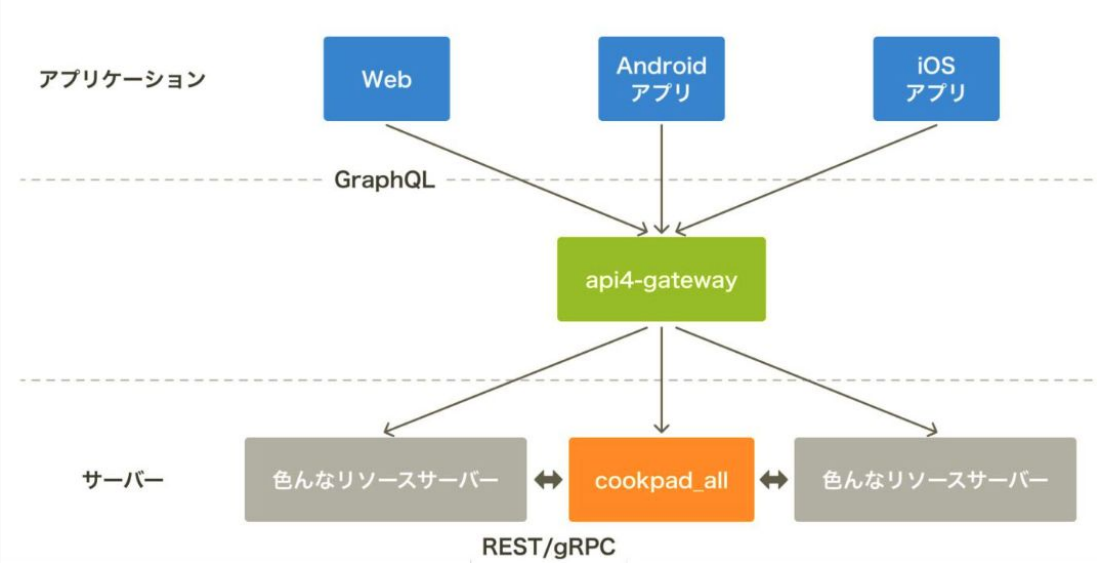
BFF のつらさ


新玉ねぎ
新着
人気
みんなのおすすめ！新玉ねぎのタグ
＋ #あと一品
＋ #春の味
＋ #お酒に合う
＋ #レンジで楽ちん
＋ #辛くない
まるごと新玉ねぎのごま豆乳鍋
丸ごと♪新玉ねぎのコンソメスープ♪
サラダ
グリル
肉じゃが
新玉ねぎとツナの和風マヨサラダ
あなたの保存レシピから 14品
☆ケトジェニック☆鶏グリル(プロテイン)
by ❀リオちゃん
リソースの結合やロジックが発生
BFF
レシピ検索用のクエリを作る
レシピ検索
マイフォルダ検索
マイフォルダのレシピ情報取得
ユーザーが有料会員？
検索結果
レシピサーバー
マイフォルダサーバー
ユーザーサーバー

何も分からん

api4-gateway の導入

- BFF のつらみを解決するために、今 Orcha, next-cookpad-api のリプレイス作業を進めている。
- GraphQL を話す**単一の** 薄い API Gateway レイヤー



GraphQL

GraphQL とは

- クライアント・サーバーがデータをやり取りするときのインターフェースの仕様
- Facebook 発で 2015 年に OSS 化
- 特徴
  - クエリ言語を使用してクライアントがほしいデータを指定して取得できる
  - Schema による型サポート、コード自動生成

なんで GraphQL？

GraphQL - REST と比較してみる

- HotRecipe（話題のレシピ）とそのつくれぽを3件取ってくる
- REST の場合と GraphQL の場合を比較してみる

GraphQL - RESTと比較

	REST	GraphQL
レスポンスの形式	返ってくるまで不明	クエリから推測可能
リクエスト回数	欲しいデータのために 複数回リクエストする必要がある	1回のリクエストで 欲しいデータを取得できる
不要なデータ	返ってくる	必要なデータのみ取得できる

※ REST の場合でも OpenAPI や Garage のようにできないことを補填する方法はある
※ gRPC の場合、レスポンスの形式を規定することはできる

なんで GraphQL？

- API の柔軟性を担保したい
- API の仕様をクライアントと共有したい

API の柔軟性を担保したい

- 汎用 API は大変
  - クライアントの特性によって必要なデータが違う
  - クライアントごとの対応でコードが複雑化
  - 各クライアント用のエンドポイントが必要になったりする
- REST API では必要なデータを取得するためのリクエスト数が多い
  - リソースごとにリクエストが必要
  - クライアントがリクエストを何回もしてそれを結合して
- クライアントの画面特化のエンドポイントを作ると、変化し続ける画面に合わせてAPI も変更しないといけない
  - UI の微修正でも API の変更が必要

API の仕様をクライアントと共有したい

- クライアントとサーバーで通信の仕様を共有したい
  - これがないとお互いに何をリクエスト・レスポンスすればいいかわからない
- nullable / non-null を定義し間違えたりするとアプリがクラッシュする恐れがある
- REST だと仕様書を見ながらレスポンスをパースするために型を写経する

GraphQL が提供したこと

- クライアントが必要なデータを選択して取得できる
- Schema に利用可能な型や操作を定義できる

クライアントがデータを選択できる

- 一回のリクエストで必要なデータを選んで取ってくることができる
- クライアント、API 双方の柔軟性が向上
  - クライアント：UI の微修正に合わせて API の変更が不要に
  - API： ユースケース特化のエンドポイントを用意しなくていい

Schema に利用可能な型や操作を定義できる

- クライアントはどういうデータや操作が利用可能なのかを知ることができる
  - ドキュメントとして利用可能
- Schema を元に、リクエスト・レスポンスの値を検証可能
- クライアントコードの自動生成にも利用可能
- その他の選択肢
  - OpenAPI https://spec.openapis.org/
  - gRPC https://grpc.io/

GraphQL の難しさ

- 従来の技術スタックが適用しづらいケースもある
  - ログやメトリクスの収集
  - キャッシュ戦略
  - エラーのセマンティクス
- パフォーマンス問題への考慮が必要
  - 柔軟なクエリによる N+1 問題の対策
  - クエリの文字列がリクエストに乗ることでネットワークコストがかかる

クックパッドにおける GraphQL

- GraphQL はまだ発展途上
  - REST からの移行が完了しきっていない
  - 運用上の課題
    - エラーハンドリングやメトリクス取得など
- マイクロサービス間通信は gRPC や REST

GraphQL の基本機能

- Schema
- Query/Mutation

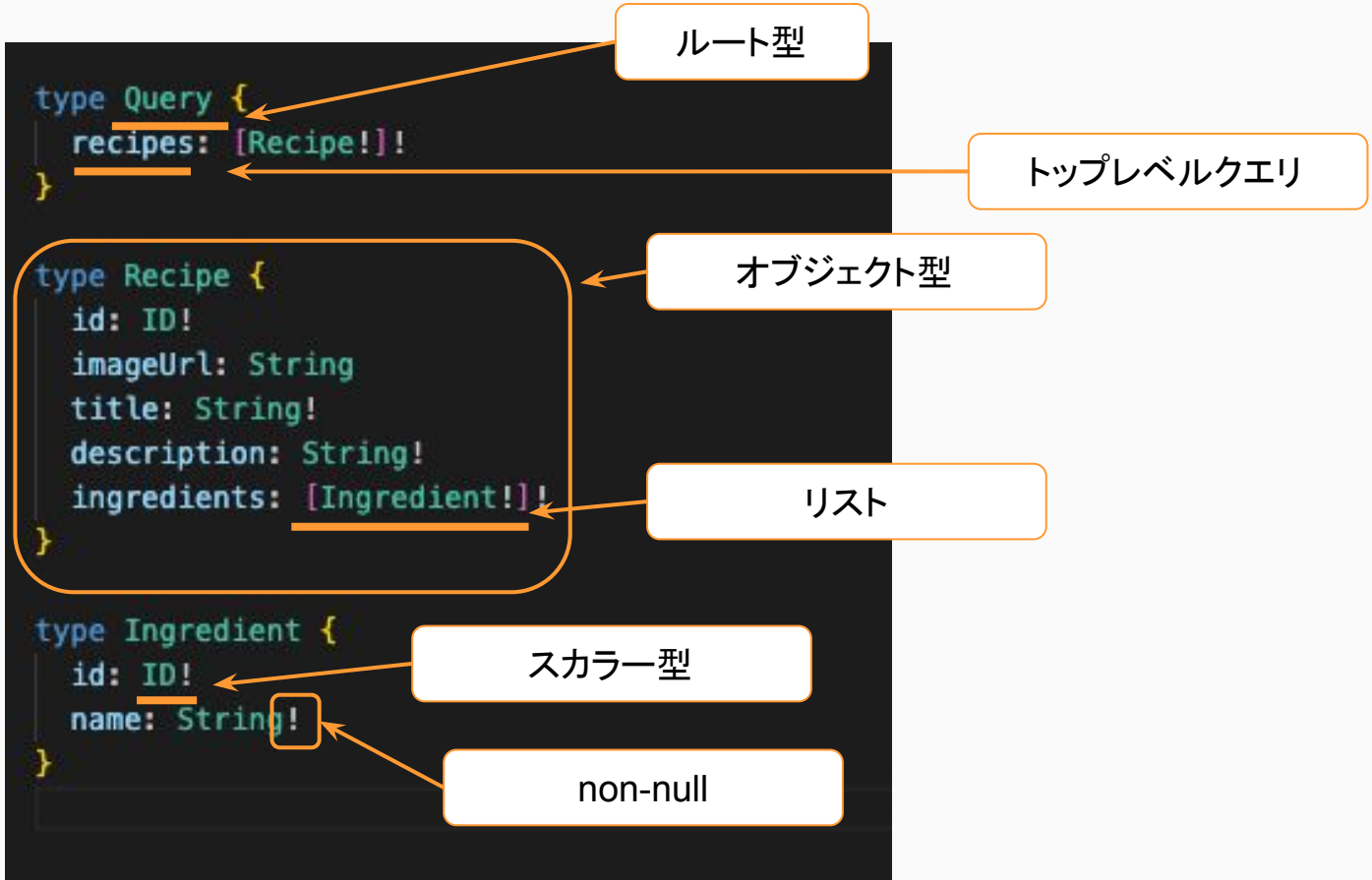
GraphQL の基本機能 - Schema

- GraphQL の型と操作を定義しているファイル
- 用途
  - リクエスト・レスポンスの値の検証
  - クライアントのコードの自動生成
- 手で書く場合と、コードを元に生成する場合がある
  - 講義で利用する graphql-ruby はコードから Schema ファイルを生成する

GraphQL の基本機能 - Schema

```
type Query {
  recipes: [Recipe!]!
}

type Recipe {
  id: ID!
  imageUrl: String
  title: String!
  description: String!
  ingredients: [Ingredient!]!
}

type Ingredient {
  id: ID!
  name: String!
}
```

GraphQL の基本機能 - Schema

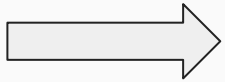


GraphQL の基本機能 - Query

- データを取得するための操作を定義
- クライアントがほしいデータを宣言的に記述し、取得することができる

Query

```
query getRecipes {
  recipes {
    title
    ingredients {
      name
    }
  }
}
```



Response


```
{
  "data": {
    "recipes": [
      {
        "title": "バレンタインに☆ココアのシンプルクッキー",
        "ingredients": [
          {
            "name": "薄力粉"
          },
          {
            "name": "純ココア"
          },
          {
            "name": "粉砂糖"
          },
```

GraphQL の基本機能 - Mutation

- データに副作用を与える操作を定義
  - 作成、更新、削除など
- RESTでいうと POST/PUT/DELETE にあたる
  - Query は GET にあたる

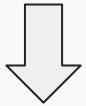
GraphQL の基本機能 - Mutation

Mutation Schema

```
type Mutation {
  addHashtags(input: AddHashtagsInput!): AddHashtagsPayload
}

input AddHashtagsInput {
  clientMutationId: String
  recipeId: ID!
  value: String!
}

type AddHashtagsPayload {
  clientMutationId: String
  hashtags: [Hashtag!]
}

type Hashtag {
  id: ID!
  name: String!
}
```

Mutation Query

```
mutation addHashtags {
  addHashtags(input: {recipeId: "7258692", value: "#平日のお昼に #リピート決定"}) {
    hashtags {
      id
      name
    }
  }
}
```

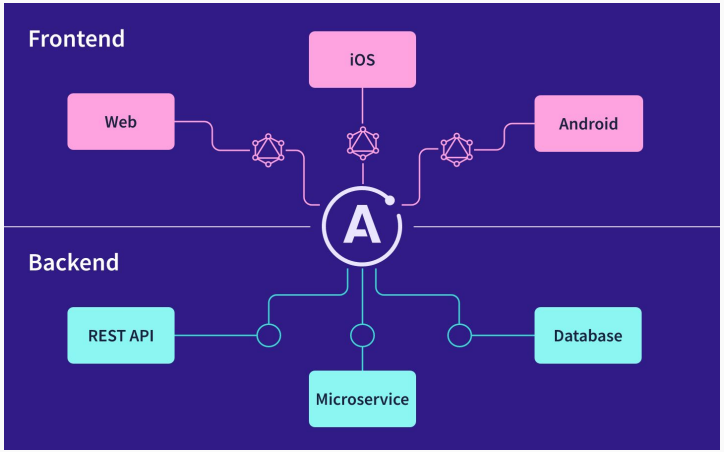


Response

```
{
  "data": {
    "addHashtags": {
      "hashtags": [
        {
          "id": "254",
          "name": "平日のお昼に"
        },
        {
```

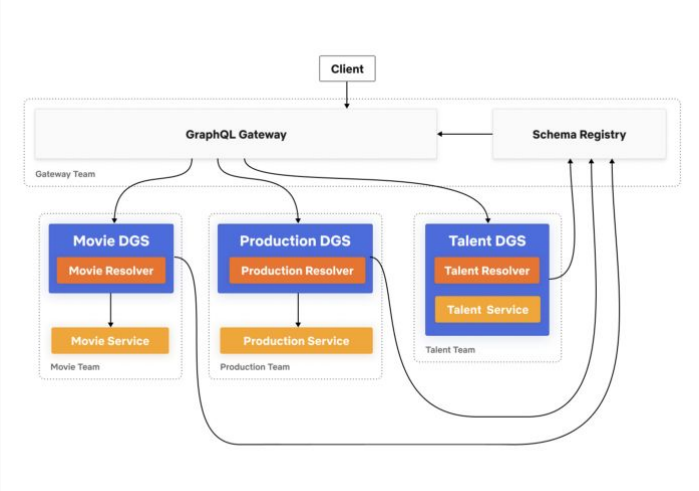
BFF と GraphQL

- Universal BFF
  - BFFの共通化
  - BFFの数が増えるにつれて重複した実装が増える課題感



https://www.apollographql.com/docs/apollo-server/ から引用

- GraphQL Federation
  - BFFにロジックをかかずに設定だけでカバー
  - マイクロサービスが増えるにつれて BFF の実装が肥大化する課題感



https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2 から引用

ということで

抜き打ちクイズ２
をします

GraphQL 理解度クイズ①

GraphQL においてレスポンスに含まれるデータを

クライアントは取捨選択できず、不要なフィールドも含まれてしまう。

マルかバツか。

GraphQL 理解度クイズ①（解答）

バツ

GraphQL においてはクライアントが必要なデータを指定して
取得できるので、ちゃんと指定すれば不要なデータは返ってこない。

GraphQL 理解度クイズ②

GraphQL においてはリクエスト・レスポンスの型を Schema で
宣言するため、クライアントとサーバー間で API の仕様を
簡単に共有できる

マルかバツか。

GraphQL 理解度クイズ②（解答）

マル

GraphQL においてはリクエスト・レスポンスの型を Schema で
宣言するため、クライアントとサーバー間で API の仕様を
簡単に共有できる

GraphQL まとめ

- GraphQL は柔軟なリクエストと API 定義の共有を実現する。
- Schema で API・データの型を宣言する。
- Query は GET、Mutation は POST/PUT/DELETE などの更新操作に対応している。

Ruby

Ruby の特徴

- すべてがオブジェクト
  - 整数も Integer というクラスのインスタンス
    - `42.class => Integer`
    - `42.times { puts "Hello World!" }`
- 動的型付け
  - Ruby 3.0 から静的型解析の仕組みも導入
- 非常に柔軟
  - やろうと思えば Ruby のコアな部分さえ変更可能

https://www.ruby-lang.org/ja/about/

クックパッドにおける Ruby

- レシピサービスや cookpad mart など全社的に利用
- マイクロサービス化で様々な言語が適材適所で利用されているが、ユーザー向けサービスだと Ruby が多い

ライブラリ

- gem と呼ばれる
  - `gem install xxx` でインストールできる
- bundler というツールで依存関係を管理
  - Gemfile に利用する gem を記載
  - `bundle install` で依存関係を解決してインストール
  - Gemfile.lock で利用しているバージョンを固定
- `bundle exec xxx` で bundler で管理している gem を利用して、コマンドを実行できる

Rails (Ruby on Rails)

Rails の特徴

- Ruby 製の Web アプリケーションフレームワーク
- Rails Way という Rails のおすすめ開発方法を提供
  - Rails というレールに乗れば、生産性が向上する
  - レールから外れると、大変なことが多い
- Rails の規約
  - 繰り返しを避けよ（Don't Repeat Yourself: DRY）
  - 設定より規約が優先（Convention Over Configuration）
- MVC パターン

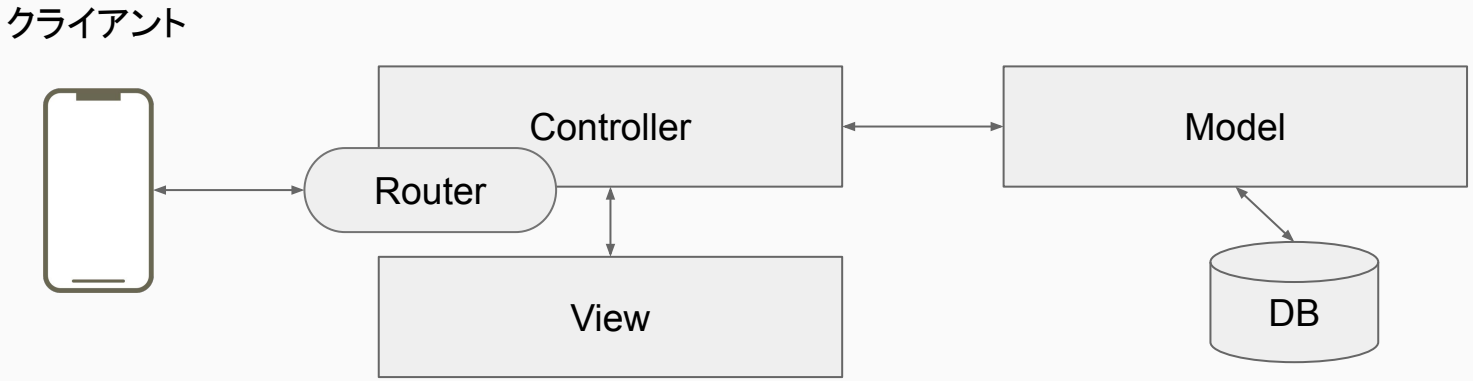
https://guides.rubyonrails.org/getting_started.html

MVC パターン

- ソフトウェアを処理 (M)、表示 (V)、入力伝達 (C) の3つに責務を分割するパターン
- Model
  - データと手続き（ビジネスロジック）を表現する要素
- View
  - モデルのデータを利用して、ユーザーに向けて描画する（Webの場合は HTML を生成する）
- Controller
  - リクエスト（入力）を受けて、モデルやビューを呼び出す

Rails の処理の流れ

1. クライアントがリクエストする
2. URI に基づいて Router がどの Controller を呼び出すかを判断する
3. Controller がリクエストされた内容に基づいて、Model や View を呼び出す
    a. Model は紐づく DB からデータを取得したり、データの処理を行う
    b. View は与えられた Model を利用して、クライアントに表示する内容を組み立てる
4. Controller が View が組み立てた結果をクライアントに返す



講義パートここまで

- ハンズオン
  - Ruby on Rails + GraphQL のアプリケーション作り
- 実際に使ってみて理解を深めましょう！
  - Rails や GraphQL の概念の理解が重要
    - なんのためにどこにコードを書くのか
    - それぞれがどういう役割なのか
  - 文法など細かいことは本質じゃないので気にしすぎなくて大丈夫です

参考リンク

- BFF
  - https://samnewman.io/patterns/architectural/bff/
  - https://www.thoughtworks.com/insights/blog/bff-soundcloud
  - https://docs.microsoft.com/ja-jp/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern
- GraphQL
  - https://graphql.org/learn/
  - https://book.productionreadygraphql.com/
  - https://www.apollographql.com/docs/

おわり